Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Комп’ютерний практикум №1

Технології паралельних обчислень

Тема: Розробка потоків та дослідження пріоритету запуску потоків

Виконав

студент групи ІП-11:

Панченко С. В.

Перевірила:

Стеценко І.В.

Київ 2024

ЗМІСТ

1 ЗАВДАННЯ

1. Реалізуйте програму імітації руху більярдних кульок, в якій рух кожної кульки відтворюється в окремому потоці (див. презентацію «Створення та запуск потоків в java» та приклад). Спостерігайте роботу програми при збільшенні кількості кульок. Поясніть результати спостереження. Опишіть переваги потокової архітектури програм. 10 балів.
2. Модифікуйте програму так, щоб при потраплянні в «лузу» кульки зникали, а відповідний потік завершував свою роботу. Кількість кульок, яка потрапила в «лузу», має динамічно відображатись у текстовому полі інтерфейсу програми. 10 балів.
3. Виконайте дослідження параметру priority потоку. Для цього модифікуйте програму «Більярдна кулька» так, щоб кульки червоного кольору створювались з вищим пріоритетом потоку, в якому вони виконують рух, ніж кульки синього кольору. Спостерігайте рух червоних та синіх кульок при збільшенні загальної кількості кульок. Проведіть такий експеримент. Створіть багато кульок синього кольору (з низьким пріоритетом) і одну червоного кольору, які починають рух в одному й тому ж самому місці більярдного стола, в одному й тому ж самому напрямку та з однаковою швидкістю. Спостерігайте рух кульки з більшим пріоритетом. Повторіть експеримент кілька разів, значно збільшуючи кожного разу кількість кульок синього кольору. Зробіть висновки про вплив пріоритету потоку на його роботу в залежності від загальної кількості потоків. 20 балів.
4. Побудуйте ілюстрацію методу join() класу Thread через взаємодію потоків, що відтворюють рух більярдних кульок різного кольору. Поясніть результат, який спостерігається. 10 балів.
5. Створіть два потоки, один з яких виводить на консоль символ ‘-‘, а інший – символ ‘|’. Запустіть потоки в основній програмі так, щоб вони виводили свої символи в рядок. Виведіть на консоль 100 таких рядків. Поясніть виведений результат. 10 балів. Використовуючи найпростіші методи управління потоками, добийтесь почергового виведення на

консоль символів. 15 балів.

6. Створіть клас Counter з методами increment() та decrement(), які збільшують та зменшують значення лічильника відповідно. Створіть два потоки, один з яких збільшує 100000 разів значення лічильника, а інший – зменшує 100000 разів значення лічильника. Запустіть потоки на одночасне виконання. Спостерігайте останнє значення лічильника. Поясніть результат. 10 балів. Використовуючи синхронізований доступ, добийтесь правильної роботи лічильника при одночасній роботі з ним двох і більше потоків. Опрацюйте використання таких способів синхронізації: синхронізований метод, синхронізований блок, блокування об’єкта. Порівняйте способи синхронізації. 15 балів.

2 ВИКОНАННЯ

2.1 Більярд

Головне меню програми складається з кнопок:

- Stop — для зупинки та виходу з програми;
- Red — для створення на полі червоних кульок;
- Blue — для створення на полі синіх кульок;
- Both — для створення однієї червоної кульки та певної кількості синіх кульок;
- Random — прапор, що відповідає за поведінку створення кульок та їхньої поведінки під час відбиття від країв екрану.
- Чотирьох лунок — місця, де якщо в них потрапляє кулька, то потік її завершує свою роботу.
- Ended threads — кількість завершених потоків.

Рисунок 2.1.1 - Головне меню програми.

Наприклад під час пустого прапору Random кульки створюються у лівому краю екрана на середні та прямують у протилежну сторону, чітко відбиваюсь знову наліво.

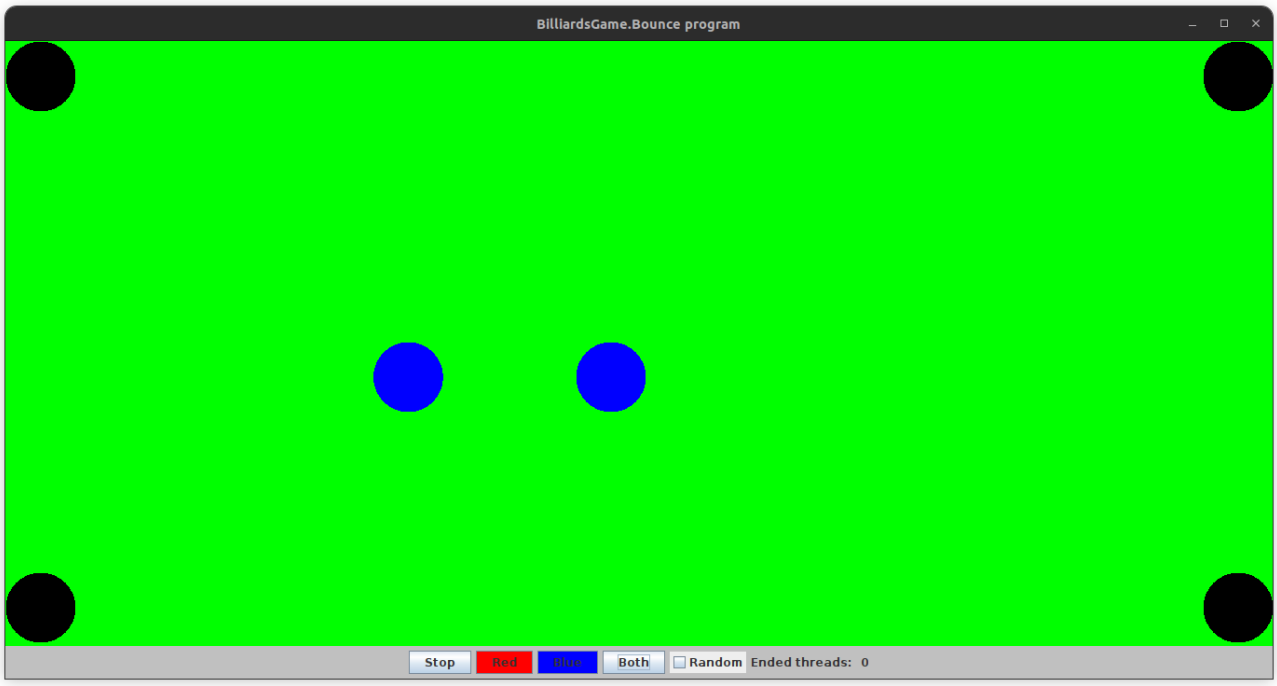

Рисунок 2.1.2 - Робота програми під час пустого прапору Random.

Продемонструємо роботу програми під час виставленого прапору Random. Побачимо, що при потраплянні в кульки поле Ended threads збільшує своє значення.

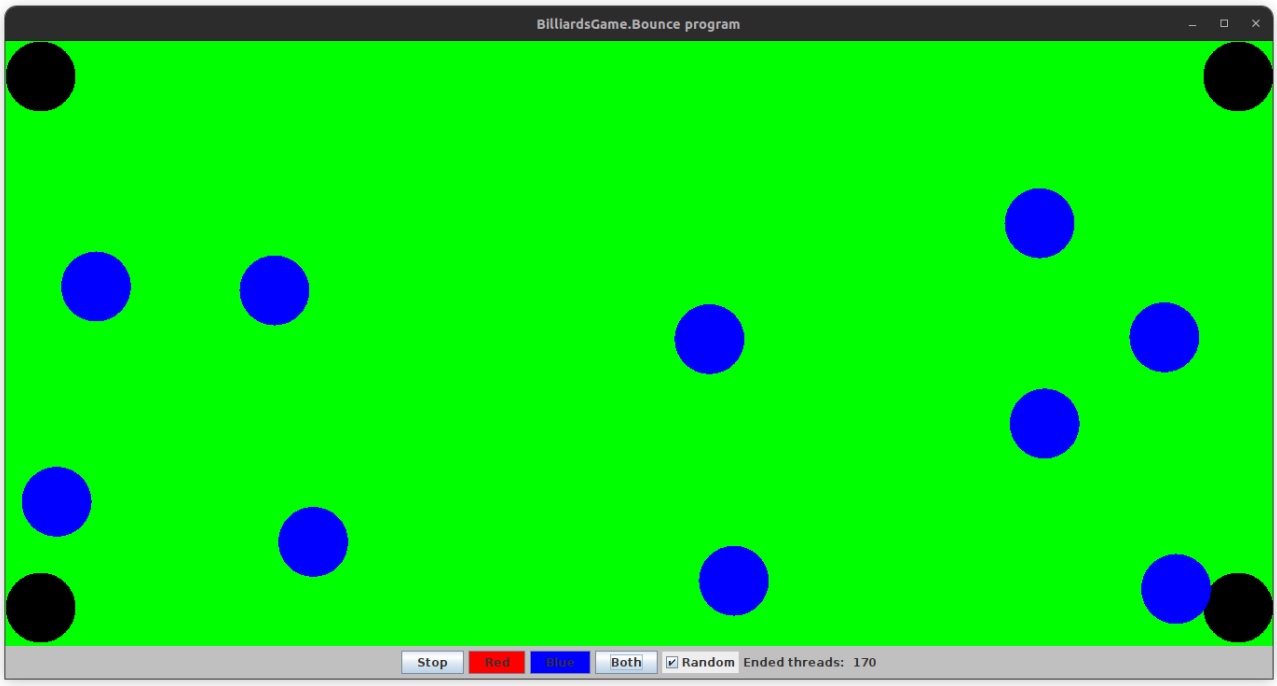

Рисунок 2.1.3 - Робота програми під час увімкненого прапору Random.

Продемонструємо пріоритетність потоку та побачимо, що попри однакову початкову швидкість та положення, червона кулька відобразилася все

таки, бо пріоритетність її потоку більша ніж відповідна їй n-кількість синіх кульок. На жаль, даний ефект можна побачити при створенні великої кількості потоків, і на даному рисунку червона кулька перекрита синіми, однак це ті сині кульки, які були створені пізніше із іншою відповідною їм червоною кулькою.

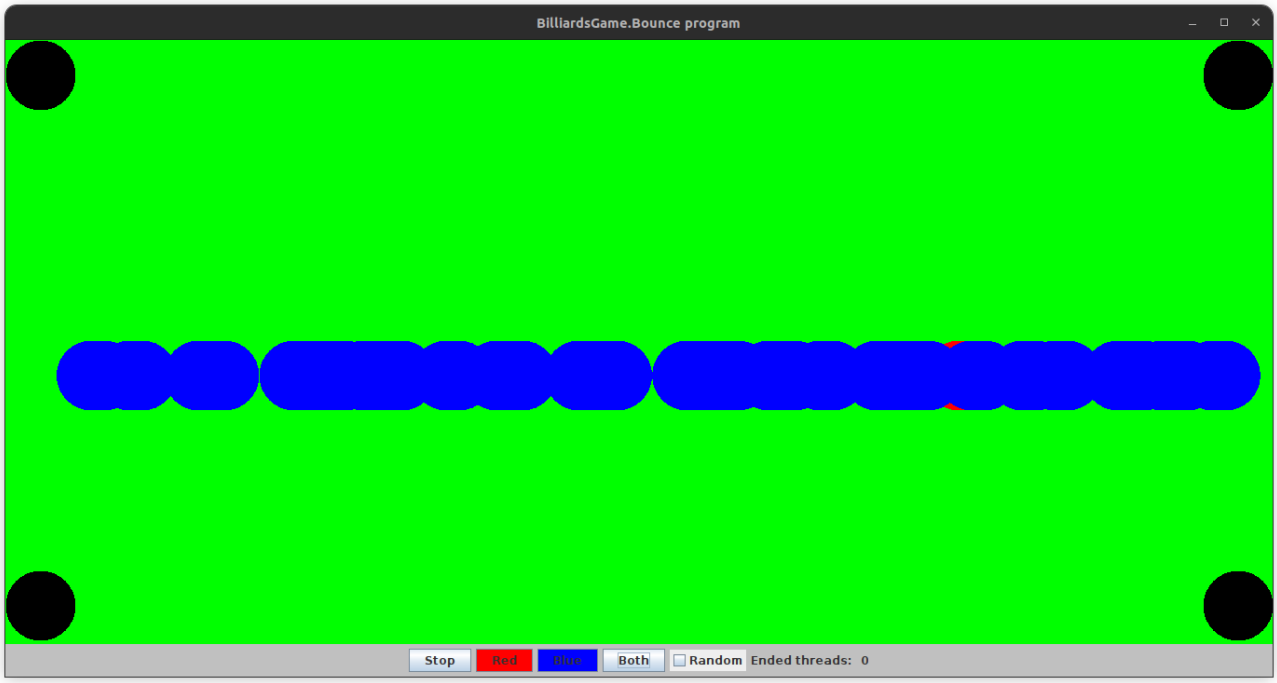

Рисунок 2.1.4 - Підтвердження роботи пріоритетності потоків.

Програма також веде логування своєї роботи. Кожен рядок містить інформацію про номер потоку; швидкість, колір, положення кульки в даний момент часу. Також є повідомлення про завершення потоку.

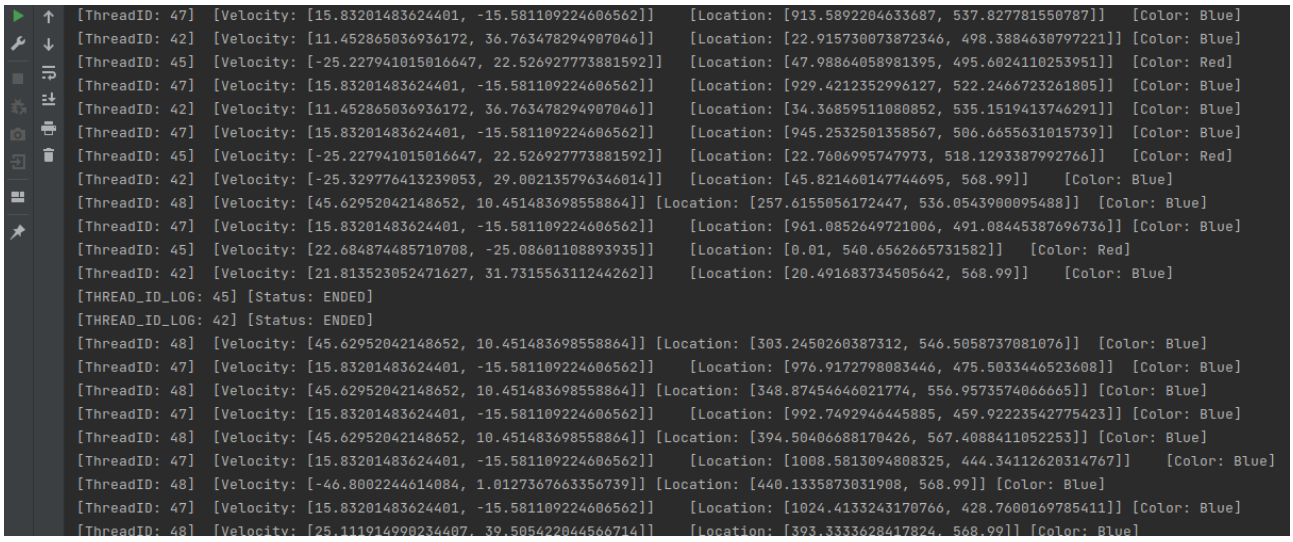

Рисунок 2.1.5 - Логування програми

Загалом структура програми реалізовує паттерн MVC(Model — View —

Control). Поглянемо на структуру.

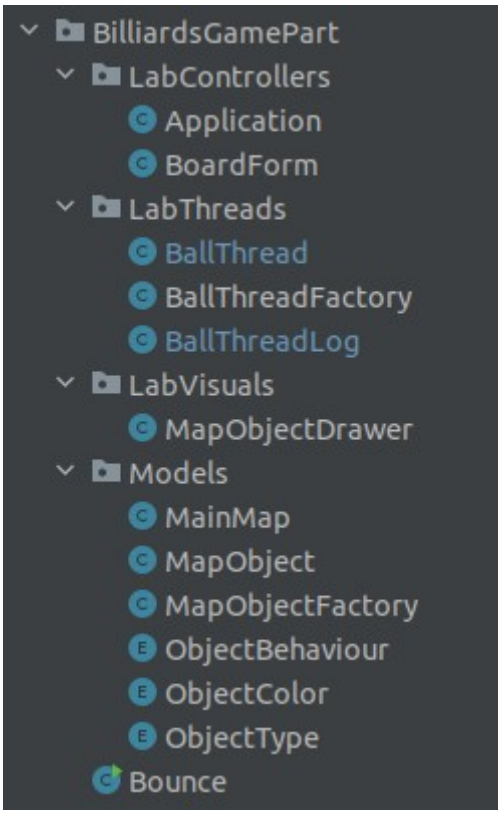

Рисунок 2.1.6 - Структура програми

Розглянемо її почергово:

1) Models — моделі об’єктів та їхні фабрики.
   - MainMap — Singleton клас, що включає в себе усі об’єкти карти.
   - MapObject — клас кульок, що перебувають на карті.
   - MapObjectFactory — фабрика кульок для контролю їхнього створення, бо загалом це залежить від поведінки кульки.
   - ObjectBehaviour — enum поведінок кульок.
   - ObjectColor — enum кольору кульок.
   - ObjectType — enum типу кульок.
2) LabVisuals — класи, що відповідають за відображення об’єктів на екран, у даному випадку кульок.
   - MapObjectDrawer — клас, що відповідає за малювання еліпсів на полотно.
3) LabControllers — форми для контролю моделей та передачі інформації на полотно.
   - BoardForm — форма, де міститься саме полотно.
   - Application — клас застосунку.

4) LabThreads — класи, що відповідають за потоки.
   - BallThread — клас, що наслідується від Thread та містить в собі посилання на відповідну кульку.
   - BallThreadFactory — фабрика, що відповідає за створення потоку, пріоритетність якого залежить від кольору кульки.
   - BallThreadLog — клас, що відповідає за логування роботи потоку.
5) Bounce — main-клас, з у якому ініціалізується об’єкт класу Application.

2.2 Виведення символів

Продемонструємо несинхронне виведення символів.

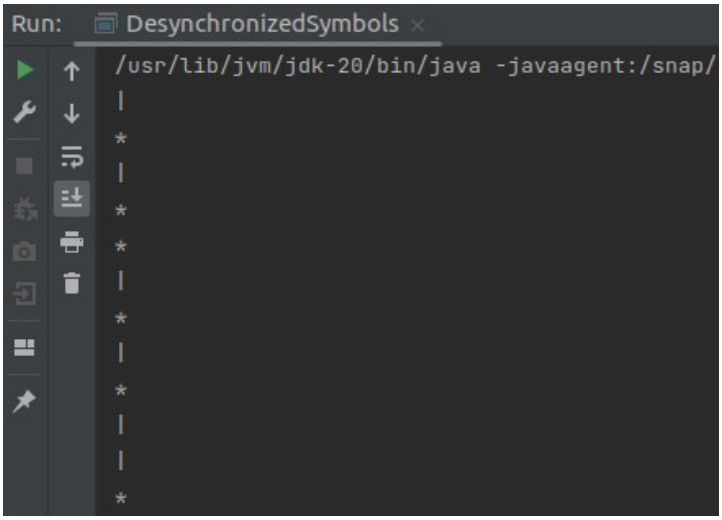

Рисунок 2.2.1 - Несинхронне виведення символів.

Продемонструємо синхронне виведення символів.

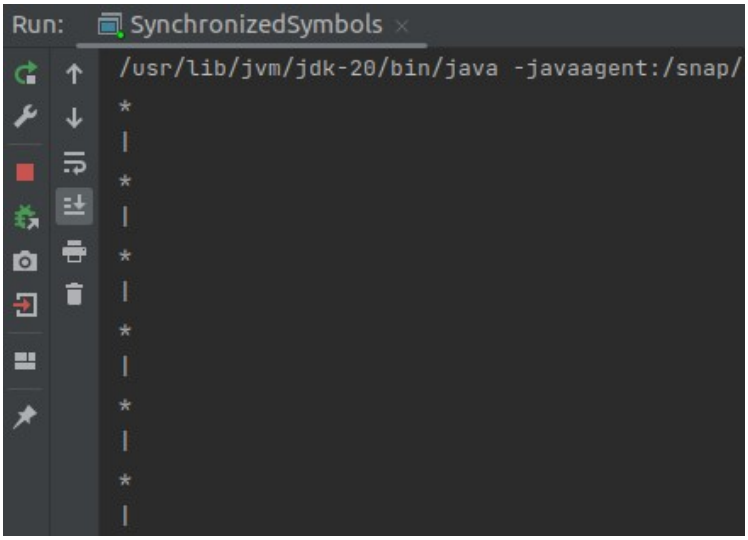

Рисунок 2.2.2 - Синхронне виведення символів.

Покажемо структуру проєкту.

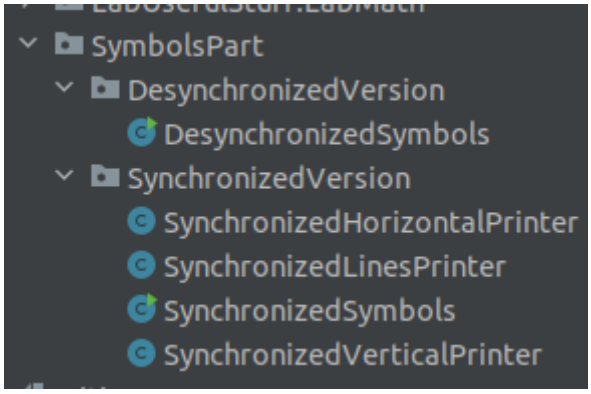

Рисунок 2.2.3 - Структура проєкту.

Розглянемо її почергово:

1) DesychronizedVersion — модуль, що містить клас, який відображає символи несинхронізовано.
   - DesychronizedSymbols — клас, який показує несинхронне зображення символів.
2) SynchronizedVersion — модуль, що відповідає за синхронне почергове відображення символів.
   - SynchronizedLinesPrinter — клас, який має синхронізовані методи класу для відображення горизонтальної та вертикальної рисок.
   - SynchronizedHorizontalPrinter та SynchronizedVerticalPrinter — класи, які мають поле об’єкта класу SynchronizedLinesPrinter та викликають відповідний їм метод. Ці класи виконують блокування монітора на даному полі, де це поле веде комунікацію між потоками через notify, забезпечуючи синхронність.
   - SynchronizedSymbols — main-клас для показу роботи програми.

2.3 Counter

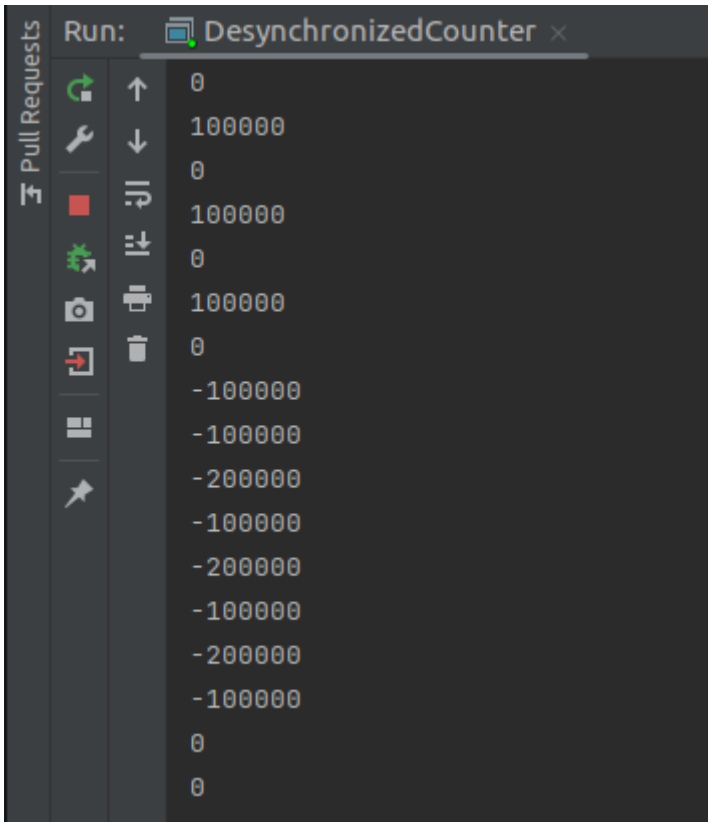

Рисунок 2.3.1 - Несинхронний підрахунок лічильника.

Оскільки три версії синхронізації лічильника видають один і той самий результат правильної синхронізації, то покажемо лиш один рисунок.

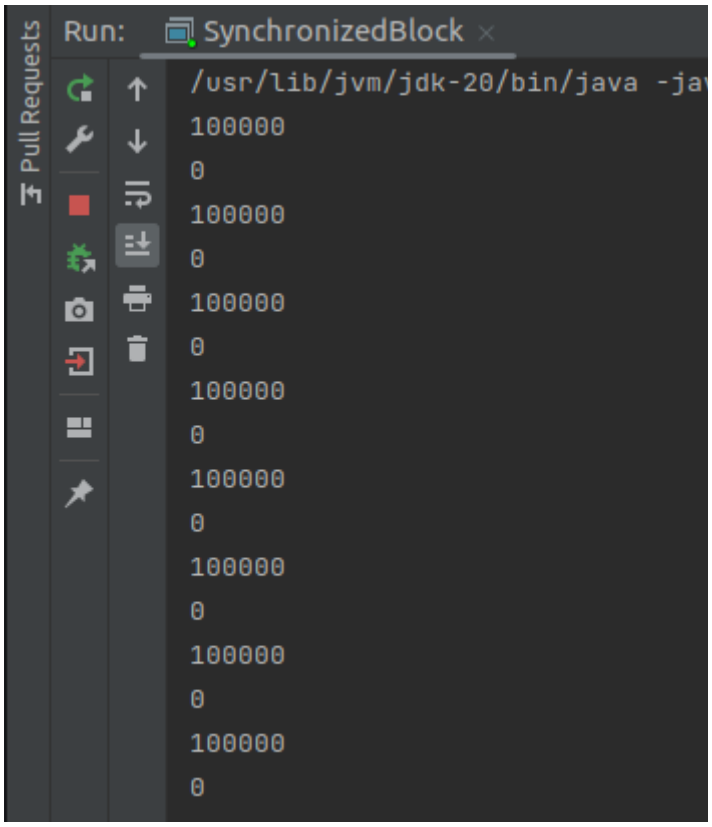

Рисунок 2.3.2 - Синхронний підрахунок лічильника.

Покажемо структуру проєкту.

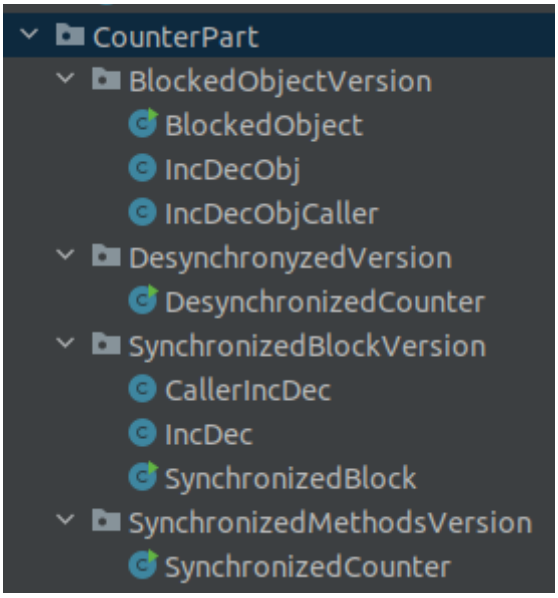

Рисунок 2.3.3 - Структура проєкту.

Розглянемо її почергово:

1) BlockedObjectVersion — модуль, що зберігає класи, які синхронізують роботу з лічильником за допомогою блокування об’єкта.
   - BlockedObject — main-клас для роботи з IncDecObj та IncDecObjCaller.
   - IncDecObj — клас, що відповідає за блокування об’єкта для синхронізації потоків.
   - IncDecObjCaller — клас, що імплементує Runnable для роботи всередині потоку.
2) DesynchronizedVersion — модуль, що зберігає клас DesynchronizedCounter, який відображає на несинхронізовану роботу з лічильником.
3) SynchronizedBlockVersion — модуль, що зберігає класи, які синхронізують роботу з лічильником за допомогою синхронізованого блоку.
   - IncDec — клас, що відповідає за синхронізацію потоків за допомогою синхронізованого блоку.
   - CallerIncDec — клас, що імплементує Runnable для роботи всередині потоку.

- SynchronizedBlock — main-клас для роботи з CallerIncDec та IncDec.

4) SynchronizedMethodsVersion — модуль, що зберігає клас SynchronizedCounter, який синхронізує роботу з лічильником за допомогою синхронізованих методів.

2.4 Математична бібліотека

Для більшої зручності створив маленьку математичну бібліотеку, яка містить в собі класи для роботи з векторами.

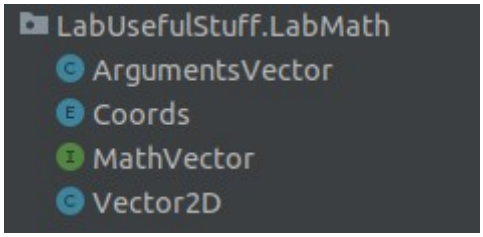

Рисунок 2.4.1 - Структура проєкту.

Розглянемо структуру:

1) MathVector — інтерфейс, що потребує від нащадків реалізації векторних операцій: додавання, множення, ділення, взяття розміру, взяття одиничного вектора тощо.
2) ArgumentsVector — нащадок MathVector, що під капотом містить звичайний масив, над яким можна виконувати векторні операції.
3) Vector2D — також нащадок MathVector, однак виконує композицію над MathVector, тобто по суті декорує його для роботи лише з двома елементами.
4) Coords — enum, який містить в собі осі x, y.

3 ВИСНОВОК

Під час лабораторної роботи опрацювали завдання з розробки програмного забезпечення з використанням різних методів синхронізації потоків: синхронізовані методи, синхронізований блок, блокування об’єкта. Однією з переваг паралельних обчислень є виграш у швидкості обробки даних та розрахунків, але зростає складність написання програми. Наприклад, під час написання більярду я стикнувся з тим, що отримував deadlock`и, оскільки не приходили сповіщення з інших потоків, і вони всі чекали один на одного.

Отже, безперечно, багатопотокові програми та методи їхньої розробки потребують вивчення, оскільки вони дають можливість виконати швидше певну дію, але на це потрібні навички та більші розумові ресурси.

ДОДАТОК А ТЕКСТИ ПРОГРАМНОГО КОДУ

Тексти програмного коду

(Найменування програми (документа))

Жорсткий диск

(Вид носія даних)

(Обсяг програми (документа), арк.)

Студента групи ІП-113 курсу

Панченка С. В

```
// ./Lab1/lab1/src/BilliardsGamePart/Bounce.java

package BilliardsGamePart;

import BilliardsGamePart.LabControllers.Application;

import javax.swing.*;

public class Bounce {
    public static void main(String[] args) {
        var frame = Application.getInstance();
        frame.init(1360, 720);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
                                System.out.println("Thread name = " +
Thread.currentThread().getName());
    }
}

// ./Lab1/lab1/src/BilliardsGamePart/LabControllers/Application.java

package BilliardsGamePart.LabControllers;

import BilliardsGamePart.LabThreads.BallThreadFactory;
import BilliardsGamePart.Models.*;

import javax.swing.*;
import java.awt.*;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;

public class Application extends JFrame implements PropertyChangeListener {
```

```
        private static final String TITLE = "BilliardsGame.Bounce program";
        private static final String STOP_TEXT = "Stop";
        private static final String RED_TEXT = "Red";
        private static final String BLUE_TEXT = "Blue";
        private static final String BOTH_TEXT = "Both";
        private static final String RANDOM_TEXT = "Random";
        private static final String ENDED_THREADS_TEXT = "Ended threads: ";
        private static final String ENDED_THREADS_COUNTER_TEXT = "0";
          private static final String IN_FRAME_THREAD_NAME = "In Frame
thread name = ";

        private static final int HOLES_COUNT = 4;
        private static final int BLUE_RED_BOTH_DIFF = 10;

        private static final double MIN_VELOCITY_PERCENTAGE = 0.01;
        private static final double MAX_VELOCITY_PERCENTAGE = 0.04;
        private static final double RADIUS_PERCENTAGE = 0.05;

        private static final int BUTTON_PANEL_HEIGHT = 76;

                private final MapObjectFactory mapObjectFactory = new
MapObjectFactory();
                private final BallThreadFactory ballThreadFactory = new
BallThreadFactory();
        private final BoardForm boardForm = new BoardForm();
                    private final JCheckBox randomCheckBox = new
JCheckBox(RANDOM_TEXT);
                     private final JLabel endedThreadsLabel = new
JLabel(ENDED_THREADS_TEXT);
                    private final JLabel joinedThreadsCounterLabel = new
JLabel(ENDED_THREADS_COUNTER_TEXT);
```

```
    private final JPanel buttonPanel = new JPanel();

    private Integer joinedThreadsCounter = 0;

    private static Application instance;

    private Application() {}

    public static Application getInstance() {
        if(instance == null) {
            instance = new Application();
        }
        return instance;
    }

    public void init(int width, int height) {
        initApplicationForm(width, height);
        initLayout();
        initMainMap();
        initMapObjectFactory();
        initHoles();
                        System.out.println(IN_FRAME_THREAD_NAME    +
Thread.currentThread().getName());
    }

    void initApplicationForm(int width, int height) {
        setSize(width, height);
        setTitle(TITLE);
    }

    private void initLayout() {
        Container content = getContentPane();
```

```
    boardForm.setBackground(Color.green);
    content.add(boardForm, BorderLayout.CENTER);

    buttonPanel.setBackground(Color.lightGray);

    var buttonStop = new JButton(STOP_TEXT);
    var buttonBallRed = new JButton(RED_TEXT);
    buttonBallRed.setBackground(Color.red);
    var buttonBallBlue = new JButton(BLUE_TEXT);
    buttonBallBlue.setBackground(Color.blue);
    var buttonBallBoth = new JButton(BOTH_TEXT);

    buttonStop.addActionListener(e -> OnButtonStop());
    buttonBallRed.addActionListener(e -> OnButtonBallRed());
    buttonBallBlue.addActionListener(e -> OnButtonBallBlue());
    buttonBallBoth.addActionListener(e -> OnButtonBallBoth());

    buttonPanel.add(buttonStop);
    buttonPanel.add(buttonBallRed);
    buttonPanel.add(buttonBallBlue);
    buttonPanel.add(buttonBallBoth);
    buttonPanel.add(randomCheckBox);
    buttonPanel.add(endedThreadsLabel);
    buttonPanel.add(joinedThreadsCounterLabel);

    buttonPanel.setSize(getWidth(), BUTTON_PANEL_HEIGHT);

    content.add(buttonPanel, BorderLayout.SOUTH);
}

private void initMainMap() {
    var mainMap = MainMap.getInstance();
```

```
                mainMap.setSize(getWidth(), getHeight() -
BUTTON_PANEL_HEIGHT);
        }

        private void initHoles() {
            var mainMap = MainMap.getInstance();
            var holes = new MapObject[HOLES_COUNT];
            for(var i = 0; i < HOLES_COUNT; ++i) {
                holes[i] = mapObjectFactory.createHole();
            }
            var diam = holes[0].getDiam();
            var size = mainMap.getSize();
            var maxX = size.getX() - diam;
            var maxY = size.getY() - diam;
            holes[0].setLocation(0, 0);
            holes[1].setLocation(maxX, 0);
            holes[2].setLocation(0, maxY);
            holes[3].setLocation(maxX, maxY);
        }

        private void initMapObjectFactory() {
            var params = new MapObjectFactory.Params();
            params.minVelocityPercentage = MIN_VELOCITY_PERCENTAGE;
            params.maxVelocityPercentage = MAX_VELOCITY_PERCENTAGE;
            params.radiusPercentage = RADIUS_PERCENTAGE;
            var size = MainMap.getInstance().getSize();
            params.straightLocation.set(1, size.getY() / 2);
                        params.straightVelocity.set(size.getX() *
MIN_VELOCITY_PERCENTAGE, 0);
            mapObjectFactory.setParams(params);
        }
```

```
private ObjectBehaviour getCreationType() {
    if(randomCheckBox.isSelected()) {
        return ObjectBehaviour.Randomized;
    }
    return ObjectBehaviour.Straight;
}

private void createThread(ObjectColor color) {
    var creationType = getCreationType();
    var ball = mapObjectFactory
            .createBall(color, creationType);
    var t = ballThreadFactory.createBallThread(ball);
    t.start();
}

private void createRedThread() {
    createThread(ObjectColor.Red);
}

private void createBlueThread() {
    createThread(ObjectColor.Blue);
}

private void OnButtonStop() {
    System.exit(0);
}

private void OnButtonBallRed() {
    createRedThread();
}

private void OnButtonBallBlue() {
```

```
        createBlueThread();
    }

    private void OnButtonBallBoth() {
        createRedThread();
        for(var i = 0; i < BLUE_RED_BOTH_DIFF; ++i) {
            createBlueThread();
        }
    }

    private synchronized void incrementJoinedThreadsLabel() {
        joinedThreadsCounter++;
        joinedThreadsCounterLabel.setText(joinedThreadsCounter.toString());
    }

    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        incrementJoinedThreadsLabel();
    }
}

// ./Lab1/lab1/src/BilliardsGamePart/LabControllers/BoardForm.java

package BilliardsGamePart.LabControllers;

import BilliardsGamePart.Models.MainMap;
import BilliardsGamePart.LabVisuals.MapObjectDrawer;
import BilliardsGamePart.Models.MapObject;

import javax.swing.*;
import java.awt.*;
```

```
public class BoardForm extends JPanel {
    public BoardForm() {}

    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Graphics2D g2 = (Graphics2D) g;
        for(MapObject b : MainMap.getInstance().getBalls()) {
            MapObjectDrawer.draw(g2, b);
        }
        repaint();
    }
}


// ./Lab1/lab1/src/BilliardsGamePart/LabThreads/BallThreadFactory.java

package BilliardsGamePart.LabThreads;

import BilliardsGamePart.Models.MapObject;
import BilliardsGamePart.Models.ObjectType;

public class BallThreadFactory {
    private static final String ERROR_NOT_BALL_TYPE = "Map object is not
ball type";
    public BallThreadFactory() {}

    public BallThread createBallThread(MapObject ball) {
                                assert  ball.getType().equals(ObjectType.Ball)  :
ERROR_NOT_BALL_TYPE;
        BallThread thread = new BallThread(ball);
        updatePriority(thread, ball);
```

```
        return thread;
    }

    public void updatePriority(BallThread thread, MapObject ball) {
        var priority = 0;
        switch(ball.getColor()) {
            case Red -> priority = Thread.MAX_PRIORITY;
            default -> priority = Thread.MIN_PRIORITY;
        }
        thread.setPriority(priority);
    }

}


// ./Lab1/lab1/src/BilliardsGamePart/LabThreads/BallThreadLog.java

package BilliardsGamePart.LabThreads;

import BilliardsGamePart.Models.MapObject;

import java.util.HashMap;

public class BallThreadLog {
    private static final String THREAD_ID_LOG = "ThreadID";
    private static final String LOCATION_LOG = "Location";
    private static final String VELOCITY_LOG = "Velocity";
    private static final String COLOR_LOG = "Color";
    private static final int LOG_CAPACITY = 200;
    private static final int LOG_HASH_CAPACITY = 4;

            private final HashMap<String, Object> logHash = new
```

```
HashMap<>(LOG_HASH_CAPACITY);
                    private final StringBuilder stringBuilder = new
StringBuilder(LOG_CAPACITY);

    private final MapObject mapObject;

    BallThreadLog(BallThread thread, MapObject inMapObject) {
      mapObject = inMapObject;
      logHash.put(THREAD_ID_LOG, thread.threadId());
      logHash.put(LOCATION_LOG, mapObject.getLocation());
      logHash.put(VELOCITY_LOG, mapObject.getVelocity());
      logHash.put(COLOR_LOG, mapObject.getColor());
    }

    public void print() {
      update();
      stringBuilder.setLength(0);
      for(var entry : logHash.entrySet()) {
        stringBuilder.append("[").append(entry.getKey()).append(": ")
            .append(entry.getValue().toString()).append("]\t");
      }
      System.out.println(stringBuilder.toString());
    }

    public void informEnded() {
                        System.out.println("[THREAD_ID_LOG:  "  +
logHash.get(THREAD_ID_LOG) + "]\t[Status: ENDED]");
    }

    private void update() {
      logHash.replace(LOCATION_LOG, mapObject.getLocation());
      logHash.replace(VELOCITY_LOG, mapObject.getVelocity());
```

```
            logHash.replace(COLOR_LOG, mapObject.getColor());
        }
    }


// ./Lab1/lab1/src/BilliardsGamePart/LabThreads/BallThread.java

package BilliardsGamePart.LabThreads;

import BilliardsGamePart.LabControllers.Application;
import BilliardsGamePart.Models.MainMap;
import BilliardsGamePart.Models.MapObject;

import java.beans.PropertyChangeSupport;

public class BallThread extends Thread {
    private static final String FINISHED = "Thread finished: ";
    private static final String STATE_PROPERTY = "State";
    private static final int SLEEP_TIME = 10;
    private final MapObject mapObject;
    private final BallThreadLog log;
    private PropertyChangeSupport support;

    public BallThread(MapObject inMapObject) {
        mapObject = inMapObject;
        log = new BallThreadLog(this, mapObject);
        support = new PropertyChangeSupport(this);
        support.addPropertyChangeListener(Application.getInstance());
    }

    @Override
    public void run() {
```

```
            try {
                while(true) {
                    mapObject.tick();
                    log.print();
                    checkValid();
                    Thread.sleep(SLEEP_TIME);
                }
            } catch(InterruptedException ignored) {
            }
        }

        private void checkValid() throws InterruptedException {
            if(MainMap.getInstance().contains(mapObject)) return;
                support.firePropertyChange(STATE_PROPERTY, State.RUNNABLE,
State.WAITING);
            log.informEnded();
            join();
        }

    }

    // ./Lab1/lab1/src/BilliardsGamePart/LabVisuals/MapObjectDrawer.java

    package BilliardsGamePart.LabVisuals;

    import BilliardsGamePart.Models.MapObject;

    import java.awt.*;
    import java.awt.geom.Ellipse2D;


    public class MapObjectDrawer {
```

```
    public static void draw(Graphics2D g, MapObject mapObject) {
        g.setColor(determineColor(mapObject));
        var loc = mapObject.getLocation();
        var diam = mapObject.getDiam();
        g.fill(new Ellipse2D.Double(loc.getX(), loc.getY(), diam, diam));
    }
    private static Color determineColor(MapObject mapObject) {
        return switch(mapObject.getColor()) {
            case Red -> Color.red;
            case Blue -> Color.blue;
            case Black -> Color.black;
        };
    }

}


// ./Lab1/lab1/src/BilliardsGamePart/Models/MapObject.java

package BilliardsGamePart.Models;

import LabUsefulStuff.LabMath.Coords;
import LabUsefulStuff.LabMath.Vector2D;

import java.util.Random;

public class MapObject {
    private static final double EPSILON = 0.01;
    private static final String AssertionErrorNotPositiveMessage = "Value is not
positive";

    private final Vector2D location = new Vector2D();
```

```
    private final Vector2D velocity = new Vector2D();
    private double diam = 0;
    private ObjectBehaviour behaviour = ObjectBehaviour.Straight;
    private ObjectColor color = ObjectColor.Blue;
    private ObjectType type = ObjectType.Ball;

    public MapObject() {
        MainMap.getInstance().addMapObject(this);
    }

    public synchronized void setLocation(Vector2D inLocation) {
        location.set(inLocation);
    }

    public synchronized void setLocation(double x, double y) {
        location.set(x, y);
    }
    public synchronized void setLocationAt(Coords coords, double value) {
        location.setAt(coords.ordinal(), value);
    }

    public synchronized Vector2D getLocation() { return location.clone(); }
        public synchronized double getLocationAt(Coords coords) { return
location.getAt(coords.ordinal()); }

    public synchronized void setDiam(int inRadius) {
        assert inRadius >= 0 : AssertionErrorNotPositiveMessage;
        diam = inRadius;
    }

    public synchronized double getDiam() { return diam; }
```

```
    public synchronized void setVelocity(Vector2D inVelocity) {
        velocity.set(inVelocity);
    }

    public synchronized void setVelocity(double x, double y) {
        velocity.set(x, y);
    }
    public synchronized void setVelocityAt(Coords coords, double value) {
        velocity.setAt(coords.ordinal(), value);
    }

    public synchronized Vector2D getVelocity() { return velocity.clone(); }
        public synchronized double getVelocityAt(Coords coords) { return
velocity.getAt(coords.ordinal()); }

    public synchronized ObjectColor getColor() { return color; }

      public synchronized void setColor(ObjectColor inObjectColor) { color =
inObjectColor; }

    public synchronized ObjectType getType() { return type; }

    public synchronized void setType(ObjectType inType) { type = inType; }

    public synchronized ObjectBehaviour getBehaviour() { return behaviour; }

        public synchronized void setBehaviour(ObjectBehaviour inBehaviour)
{ behaviour = inBehaviour; }

    public synchronized Vector2D getCenterLocation() {
        var center = new Vector2D();
        center.set(location.getX() + diam, location.getY() + diam);
```

```
    return center;
}

public synchronized void tick() {
    location.add(velocity);
    if(isInsideHole()) {
        MainMap.getInstance().removeMapObject(this);
    }
    bounceMap();
}

private synchronized void bounceMap() {
    checkBounce(Coords.X);
    checkBounce(Coords.Y);
}

private synchronized void rotateVelocityRandomly() {
    var forwardVec = velocity.getForwardVector();
    var deg = (new Random()).nextDouble(360);
    forwardVec.rotate(deg);
    forwardVec.multiply(velocity.getSize());
    setVelocity(forwardVec);
}

private synchronized boolean isPointInside(Vector2D point) {
    var dist = getCenterLocation().getDistance(point);
    var res = (dist - getDiam() / 2) < EPSILON;
    return res;
}

private synchronized boolean isInsideHole() {
    for(var obj : MainMap.getInstance().getBalls()) {
```

```
            if(obj.type.equals(ObjectType.Hole)) {
                var res = obj.isPointInside(getCenterLocation());
                if(res) {
                    return true;
                }
            }
        }
        return false;
    }

    private synchronized void checkBounce(Coords coords) {
                                                    var maxCoord =
MainMap.getInstance().getSize().getAt(coords.ordinal());
        var curCoord = getLocationAt(coords);
        var isBounced = false;
        if(curCoord < 0) {
            setLocationAt(coords, EPSILON);
            isBounced = true;
        } else if(curCoord + diam - maxCoord > EPSILON) {
            setLocationAt(coords, maxCoord - diam - EPSILON);
            isBounced = true;
        }
        if(isBounced) {
            switch(getBehaviour()) {
                case Straight -> setVelocityAt(coords, -getVelocityAt(coords));
                case Randomized -> rotateVelocityRandomly();
            }
        }
    }
}
```

```
// ./Lab1/lab1/src/BilliardsGamePart/Models/MapObjectFactory.java

package BilliardsGamePart.Models;

import LabUsefulStuff.LabMath.Vector2D;

import java.util.Random;

public class MapObjectFactory {
    private final Random random = new Random();
    private Params params;

    public static class Params implements Cloneable {
        public double minVelocityPercentage = 0;
        public double maxVelocityPercentage = 0;
        public double radiusPercentage = 0;
        public final Vector2D straightLocation = new Vector2D();
        public final Vector2D straightVelocity = new Vector2D();

        @Override
        public Params clone() {
            try {
                return (Params) super.clone();
            } catch (CloneNotSupportedException exception) {
                exception.printStackTrace();
            }
            return null;
        }
    }

    public MapObjectFactory() {}
```

```
    public void setParams(Params inParams) {
        params = inParams.clone();
        checkParams();
    }

    public MapObject createHole() {
        checkParams();
        var mapObject = new MapObject();
        mapObject.setType(ObjectType.Hole);
        mapObject.setColor(ObjectColor.Black);
        updateRadius(mapObject);
        return mapObject;
    }

        public MapObject createBall(ObjectColor color, ObjectBehaviour
behaviour) {
        checkParams();
        var mapObject = new MapObject();
        mapObject.setBehaviour(behaviour);
        mapObject.setType(ObjectType.Ball);
        mapObject.setColor(color);
        updateRadius(mapObject);
        updateLocation(mapObject);
        updateVelocity(mapObject);
        return mapObject;
    }


    private void checkParams() {
        assert params.radiusPercentage >= 0 && params.radiusPercentage <= 1;
                        assert  params.minVelocityPercentage  >=  0  &&
```

```
params.minVelocityPercentage <= 1;
                assert params.maxVelocityPercentage >= 0 &&
params.maxVelocityPercentage <= 1;
        assert params.minVelocityPercentage <= params.maxVelocityPercentage;
    }

    private void updateRadius(MapObject mapObject) {
        var mapSize = MainMap.getInstance().getSize();
            var magnitude = Math.sqrt(Math.pow(mapSize.getX(), 2) +
Math.pow(mapSize.getY(), 2));
        var radius = (int)(params.radiusPercentage * magnitude);
        mapObject.setDiam(radius);
    }

    private void updateLocation(MapObject mapObject) {
        switch(mapObject.getBehaviour()) {
            case Straight -> updateLocationStraight(mapObject);
            case Randomized -> updateLocationRandomized(mapObject);
        }
    }

    private void updateLocationStraight(MapObject mapObject) {
        mapObject.setLocation(params.straightLocation);
    }

    private void updateLocationRandomized(MapObject mapObject) {
        var mapSize = MainMap.getInstance().getSize();
        var ballRadius = mapObject.getDiam();

        var x = random.nextDouble(mapSize.getX() - ballRadius);
        var y = random.nextDouble(mapSize.getY() - ballRadius);
```

```
        mapObject.setLocation(x, y);
    }

    private void updateVelocity(MapObject mapObject) {
        switch(mapObject.getBehaviour()) {
            case Straight -> updateVelocityStraight(mapObject);
            case Randomized -> updateVelocityRandomized(mapObject);
        }
    }

    private void updateVelocityStraight(MapObject mapObject) {
        mapObject.setVelocity(params.straightVelocity);
    }

    private void updateVelocityRandomized(MapObject mapObject) {
        var mapSize = MainMap.getInstance().getSize();

        var minValX = mapSize.getX() * params.minVelocityPercentage;
        var maxValX = mapSize.getX() * params.maxVelocityPercentage;
        var minValY = mapSize.getY() * params.minVelocityPercentage;
        var maxValY = mapSize.getY() * params.maxVelocityPercentage;

        var vX = minValX + random.nextDouble(maxValX - minValX);
        var vY = minValY + random.nextDouble(maxValY - minValY);

        mapObject.setVelocity(vX, vY);
    }
}


// ./Lab1/lab1/src/BilliardsGamePart/Models/MainMap.java
```

```
package BilliardsGamePart.Models;

import java.util.ArrayList;

import LabUsefulStuff.LabMath.Vector2D;

public class MainMap {

    private final Vector2D size = new Vector2D();
    private final ArrayList<MapObject> mapObjects = new ArrayList<>();
    private final ArrayList<MapObject> holes = new ArrayList<>();
    private static MainMap instance;

    private MainMap() {}

    public synchronized static MainMap getInstance() {
        if(instance == null) {
            instance = new MainMap();
        }
        return instance;
    }

    public synchronized void setSize(Vector2D inSize) {
        size.set(inSize);
    }

    public synchronized void setSize(double x, double y) {
        size.set(x, y);
    }

    public synchronized Vector2D getSize() {
        return size.clone();
```

```
    }

    public synchronized ArrayList<MapObject> getBalls() {
        return (ArrayList<MapObject>) mapObjects.clone();
    }

    public synchronized void addMapObject(MapObject mapObject) {
        mapObjects.add(mapObject);
    }

    public synchronized void removeMapObject(MapObject mapObject) {
        mapObjects.remove(mapObject);
    }

    public synchronized boolean contains(MapObject mapObject) {
        return mapObjects.contains(mapObject);
    }
}


// ./Lab1/lab1/src/BilliardsGamePart/Models/ObjectBehaviour.java

package BilliardsGamePart.Models;

public enum ObjectBehaviour {
    Straight,
    Randomized
}


// ./Lab1/lab1/src/BilliardsGamePart/Models/ObjectColor.java
```

```
package BilliardsGamePart.Models;

public enum ObjectColor {
    Red,
    Blue,
    Black
}


// ./Lab1/lab1/src/BilliardsGamePart/Models/ObjectType.java

package BilliardsGamePart.Models;

public enum ObjectType {
    Ball,
    Hole
}


//
./Lab1/lab1/src/CounterPart/DesynchronyzedVersion/DesynchronizedCounter.java

package CounterPart.DesynchronyzedVersion;

public class DesynchronizedCounter {
    private static final int STEP = 100000;
    private static final int WAIT_TIME = 100;
    private int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        var desynchronizedCounter = new DesynchronizedCounter();
```

```
    var thread1 = new Thread(desynchronizedCounter::increment);
    var thread2 = new Thread(desynchronizedCounter::decrement);

    thread1.start();
    thread2.start();

    thread1.join();
    thread2.join();
}

public void increment() {
    while(true) {
        counter += STEP;
        System.out.println(counter);
        try {
            Thread.sleep(WAIT_TIME);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public void decrement() {
    while(true) {
        counter -= STEP;
        System.out.println(counter);
        try {
            Thread.sleep(WAIT_TIME);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
    }
}


//
./Lab1/lab1/src/CounterPart/SynchronizedMethodsVersion/SynchronizedCounter.jav
a

package CounterPart.SynchronizedMethodsVersion;

import java.util.Vector;

public class SynchronizedCounter {
    private static final int STEP = 100000;
    private static final int WAIT_TIME = 100;

    private static boolean semaphore = true;
    private static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        int totalIncrementingThreads = 2;
        int totalDecrementingThreads = 3;
        var threads = new Vector<Thread>();

        for(int i = 0; i < totalIncrementingThreads; i++) {
            threads.add(new Thread(() -> doWork(true)));
        }

        for(int i = 0; i < totalDecrementingThreads; i++) {
            threads.add(new Thread(() -> doWork(false)));
        }
```

```
    for(var thread : threads) {
        thread.start();
    }

    for(var thread : threads) {
        thread.join();
    }
}

private static void doWork(boolean isIncrementing) {
    while(true) {
        incDec(isIncrementing);
    }
}

private static synchronized void incDec(boolean isIncrementing) {
    if(isIncrementing && semaphore) {
        counter += STEP;
        System.out.println(counter);
        semaphore = false;
        sleep();
    } else if(!isIncrementing && !semaphore) {
        counter -= STEP;
        System.out.println(counter);
        semaphore = true;
        sleep();
    }
}

private static void sleep() {
    try {
        Thread.sleep(WAIT_TIME);
```

```
        } catch(InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}


// ./Lab1/lab1/src/CounterPart/BlockedObjectVersion/IncDecObj.java

package CounterPart.BlockedObjectVersion;

public class IncDecObj {
    private static final int WAIT_TIME = 100;
    private int counter = 0;
    private int step = 100000;
    private boolean isIncrementing = true;

    IncDecObj(int step) {
        this.step = step;
    }

    synchronized void increment() {
        while(isIncrementing) doWait();
        counter += step;
        isIncrementing = true;
        printCounter();
        sleep();
        notifyAll();
    }

    synchronized void decrement() {
        while(!isIncrementing) doWait();
```

```
            counter -= step;
            isIncrementing = false;
            printCounter();
            sleep();
            notifyAll();
        }

        private void printCounter() {
            System.out.println(counter);
        }

        void doWait() {
            try {
                wait();
            } catch(InterruptedException e) {
                e.printStackTrace();
            }
        }

        void sleep() {
            try {
                Thread.sleep(WAIT_TIME);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    // ./Lab1/lab1/src/CounterPart/BlockedObjectVersion/IncDecObjCaller.java

    package CounterPart.BlockedObjectVersion;
```

```
public class IncDecObjCaller implements Runnable {
    private IncDecObj incDecObj;
    private boolean isInc = true;

    public IncDecObjCaller(IncDecObj incDecObj, boolean isInc) {
        this.incDecObj = incDecObj;
        this.isInc = isInc;
    }

    public void run() {
        while(true) {
            if(isInc) {
                incDecObj.increment();
                continue;
            }
            incDecObj.decrement();
        }
    }
}


// ./Lab1/lab1/src/CounterPart/BlockedObjectVersion/BlockedObject.java

package CounterPart.BlockedObjectVersion;

import java.util.Vector;

public class BlockedObject {
    public static void main(String[] args) {
        var step = 100000;
        IncDecObj incDecObj = new IncDecObj(step);
```

```
        int totalIncrementingThreads = 6;
        int totalDecrementingThreads = 10;
        var threads = new Vector<Thread>();

        for(int i = 0; i < totalIncrementingThreads; i++) {
            threads.add(new Thread(new IncDecObjCaller(incDecObj, true)));
        }

        for(int i = 0; i < totalDecrementingThreads; i++) {
            threads.add(new Thread(new IncDecObjCaller(incDecObj, false)));
        }

        for(var thread : threads) {
            thread.start();
        }

        for(var thread : threads) {
            try {
                thread.join();
            } catch(InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}


// ./Lab1/lab1/src/CounterPart/SynchronizedBlockVersion/CallerIncDec.java

package CounterPart.SynchronizedBlockVersion;
```

```
public class CallerIncDec implements Runnable {
    private final IncDec incDec;
    private boolean isInc = true;

    CallerIncDec(IncDec incDec, boolean isInc) {
        this.incDec = incDec;
        this.isInc = isInc;
    }


    @Override
    public void run() {
        while(true) {
            synchronized(incDec) {
                if(isInc) {
                    incDec.increment();
                } else {
                    incDec.decrement();
                }
            }
        }
    }
}


//
./Lab1/lab1/src/CounterPart/SynchronizedBlockVersion/SynchronizedBlock.java

package CounterPart.SynchronizedBlockVersion;

import java.util.Vector;
```

```
public class SynchronizedBlock {
    public static void main(String[] args) {
        var incDec = new IncDec(100000);

        int totalIncrementingThreads = 6;
        int totalDecrementingThreads = 10;
        var threads = new Vector<Thread>();

        for(int i = 0; i < totalIncrementingThreads; i++) {
            threads.add(new Thread(new CallerIncDec(incDec, true)));
        }

        for(int i = 0; i < totalDecrementingThreads; i++) {
            threads.add(new Thread(new CallerIncDec(incDec, false)));
        }

        for(var thread : threads) {
            thread.start();
        }

        for(var thread : threads) {
            try {
                thread.join();
            } catch(InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

```
// ./Lab1/lab1/src/CounterPart/SynchronizedBlockVersion/IncDec.java

package CounterPart.SynchronizedBlockVersion;

public class IncDec {
    private static final int SLEEP_TIME = 100;

    private int step = 100000;
    private int counter = 0;
    private boolean isIncOrDec = true;

    IncDec(int step) {
        this.step = step;
    }

    public void increment() {
        if(!isIncOrDec) return;
        isIncOrDec = false;
        counter += step;
        printCounter();
        sleep();
    }

    public void decrement() {
        if(isIncOrDec) return;
        isIncOrDec = true;
        counter -= step;
        printCounter();
        sleep();
    }

    private void printCounter() {
```

```
            System.out.println(counter);
        }

        private void sleep() {
            try {
                Thread.sleep(SLEEP_TIME);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }


    //
./Lab1/lab1/src/SymbolsPart/DesynchronizedVersion/DesynchronizedSymbols.java

    package SymbolsPart.DesynchronizedVersion;

    public class DesynchronizedSymbols {
        public static void main(String[] args) throws InterruptedException {
            var counts = 100;
            var waitTime = 10;
            var thread1 = new Thread(() -> printVerticalLine(counts, waitTime));
            var thread2 = new Thread(() -> printHorizontalLine(counts, waitTime));

            thread1.start();
            thread2.start();

            thread1.join();
            thread2.join();
        }
        private static void printVerticalLine(int counts, int waitTime) {
```

```
            for (int i = 0; i < counts; i++) {
                System.out.println("|");
                try {
                    Thread.sleep(waitTime);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        private static void printHorizontalLine(int counts, int waitTime) {
            for (int i = 0; i < counts; i++) {
                System.out.println("*");
                try {
                    Thread.sleep(waitTime);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }


    //
./Lab1/lab1/src/SymbolsPart/SynchronizedVersion/SynchronizedSymbols.java

    package SymbolsPart.SynchronizedVersion;

    public class SynchronizedSymbols {
        public static void main(String[] args) throws InterruptedException {
            var linesPrinter = new SynchronizedLinesPrinter();

            var verticalPrinter = new SynchronizedVerticalPrinter(linesPrinter);
```

```
            var horizontalPrinter = new SynchronizedHorizontalPrinter(linesPrinter);

                                        var verticalPrinterThread = new
Thread(verticalPrinter::printVerticalLine);
                                        var horizontalPrinterThread = new
Thread(horizontalPrinter::printHorizontalLine);

            verticalPrinterThread.start();
            horizontalPrinterThread.start();

            verticalPrinterThread.join();
            horizontalPrinterThread.join();
        }
    }


    //
./Lab1/lab1/src/SymbolsPart/SynchronizedVersion/SynchronizedVerticalPrinter.java

    package SymbolsPart.SynchronizedVersion;

    public class SynchronizedVerticalPrinter {
        private final SynchronizedLinesPrinter synchronizedLinesPrinter;
                                SynchronizedVerticalPrinter(SynchronizedLinesPrinter
synchronizedLinesPrinter) {
            this.synchronizedLinesPrinter = synchronizedLinesPrinter;
        }
        public void printVerticalLine() {
            while(true) {
                synchronizedLinesPrinter.printVerticalLine();
            }
        }
```

```
    }


    //
./Lab1/lab1/src/SymbolsPart/SynchronizedVersion/SynchronizedLinesPrinter.java

    package SymbolsPart.SynchronizedVersion;

    public class SynchronizedLinesPrinter {
        private static final int WAIT_TIME = 1000;
        private boolean isVerticalLine = true;
        SynchronizedLinesPrinter() {}
        synchronized void printVerticalLine() {
            while(isVerticalLine) {
                try {
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("|");
            isVerticalLine = true;
            try {
                Thread.sleep(WAIT_TIME);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            notify();
        }
        synchronized void printHorizontalLine() {
            while(!isVerticalLine) {
                try {
```

```
                wait();
            } catch(InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("*");
        isVerticalLine = false;
        try {
            Thread.sleep(WAIT_TIME);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        notify();
    }
}


//
./Lab1/lab1/src/SymbolsPart/SynchronizedVersion/SynchronizedHorizontalPrinter.ja
va

package SymbolsPart.SynchronizedVersion;

public class SynchronizedHorizontalPrinter {
    private final SynchronizedLinesPrinter synchronizedLinesPrinter;
                        SynchronizedHorizontalPrinter(SynchronizedLinesPrinter
synchronizedLinesPrinter) {
        this.synchronizedLinesPrinter = synchronizedLinesPrinter;
    }
    public void printHorizontalLine() {
        while(true) {
            synchronizedLinesPrinter.printHorizontalLine();
```

```
        }
    }
}


// ./Lab1/lab1/src/LabUsefulStuff/LabMath/MathVector.java

package LabUsefulStuff.LabMath;

public interface MathVector<T> {
    void add(T other);
    void sub(T other);
    void multiply(T other);
    void multiply(double value);
    void divide(T other);
    void divide(double value);
    double getSize();
    double getSizeSquared();
    double getDotProduct(T other);
    double getDistance(T other);
    T getForwardVector();
    double getAt(int index);
    void setAt(int index, double value);
    T getOpposite();
    void toOpposite();
    void set(T other);
}


// ./Lab1/lab1/src/LabUsefulStuff/LabMath/Coords.java

package LabUsefulStuff.LabMath;
```

```
public enum Coords {
   X,
   Y
}


// ./Lab1/lab1/src/LabUsefulStuff/LabMath/ArgumentsVector.java

package LabUsefulStuff.LabMath;

import java.util.Arrays;

public class ArgumentsVector implements MathVector<ArgumentsVector> {
   private static final String ERROR_LENGTHS_NOT_EQUAL = "Lengths of
points arguments are not equal";
   private double[] arguments;

   ArgumentsVector(int length) {
      setLength(length);
   }

   ArgumentsVector(ArgumentsVector other) {
      setLength(other.getLength());
      set(other);
   }

   public int getLength() {
      if(arguments == null) {
         return 0;
      }
      return arguments.length;
```

```
}

public void setLength(int length) {
    var currentLength = getLength();

    if(currentLength==length) return;

    var minLength = Math.min(currentLength, length);
    var args = new double[length];
    for(var i = 0; i < minLength; ++i) {
        args[i] = getAt(i);
    }

    arguments = args;
}

@Override
public void set(ArgumentsVector other) {
    checkSizesEqual(other);
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, other.getAt(i));
    }
}

@Override
public ArgumentsVector clone() {
    return new ArgumentsVector(this);
}

@Override
public String toString() {
    return Arrays.toString(arguments);
```

```
    }

    private void checkSizesEqual(ArgumentsVector other) {
                                assert getLength() == other.getLength() :
ERROR_LENGTHS_NOT_EQUAL;
    }

    @Override
    public void add(ArgumentsVector other) {
       checkSizesEqual(other);
       for(var i = 0; i < getLength(); ++i) {
          setAt(i, getAt(i) + other.getAt(i));
       }
    }

    @Override
    public void sub(ArgumentsVector other) {
       checkSizesEqual(other);
       for(var i = 0; i < getLength(); ++i) {
          setAt(i, getAt(i) - other.getAt(i));
       }
    }

    @Override
    public void multiply(ArgumentsVector other) {
       checkSizesEqual(other);
       for(var i = 0; i < getLength(); ++i) {
          setAt(i, getAt(i) * other.getAt(i));
       }
    }

    @Override
```

```
public void multiply(double value) {
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, getAt(i) * value);
    }
}

@Override
public void divide(ArgumentsVector other) {
    checkSizesEqual(other);
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, getAt(i) / other.getAt(i));
    }
}

@Override
public void divide(double value) {
    for(var i = 0; i < getLength(); ++i) {
        setAt(i, getAt(i) / value);
    }
}

@Override
public double getSize() {
    return Math.sqrt(getSizeSquared());
}

@Override
public double getSizeSquared() {
    double s = 0;
    for(var i = 0; i < getLength(); ++i) {
        s += Math.pow(getAt(i), 2);
    }
```

```
    return s;
}

@Override
public double getDotProduct(ArgumentsVector other) {
    checkSizesEqual(other);
    var prod = 0;
    for(var i = 0; i < getLength(); ++i) {
        prod += getAt(i) * other.getAt(i);
    }
    return prod;
}

@Override
public double getDistance(ArgumentsVector other) {
    checkSizesEqual(other);
    var dist = 0.0;
    for(var i = 0; i < getLength(); ++i) {
        dist += Math.pow(getAt(i) - other.getAt(i), 2);
    }
    dist = Math.sqrt(dist);
    return dist;
}

@Override
public ArgumentsVector getForwardVector() {
    var forwardVec = clone();
    var size = getSize();
    for(var i = 0; i < getLength(); ++i) {
        forwardVec.setAt(i, getAt(i) / size);
    }
    return forwardVec;
```

```
    }

    @Override
    public double getAt(int index) {
        return arguments[index];
    }

    @Override
    public void setAt(int index, double value) {
        arguments[index] = value;
    }

    @Override
    public ArgumentsVector getOpposite() {
        var v = clone();
        v.toOpposite();
        return v;
    }

    @Override
    public void toOpposite() {
        for(var i = 0; i < getLength(); ++i) {
            setAt(i, -getAt(i));
        }
    }

}


// ./Lab1/lab1/src/LabUsefulStuff/LabMath/Vector2D.java

package LabUsefulStuff.LabMath;
```

```
public class Vector2D implements MathVector<Vector2D> {
    private final ArgumentsVector vec = new ArgumentsVector(2);

    public Vector2D() {}

    public Vector2D(double x, double y) {
        set(x, y);
    }

    public Vector2D(Vector2D other) {
        set(other);
    }

    public double getX() {
        return getAt(Coords.X.ordinal());
    }

    public double getY() {
        return getAt(Coords.Y.ordinal());
    }

    public void set(Vector2D other) {
        vec.set(other.vec);
    }

    public void set(double x, double y) {
        setX(x);
        setY(y);
    }

    public void setX(double value) {
```

```
    setAt(Coords.X.ordinal(), value);
}

public void setY(double value) {
    setAt(Coords.Y.ordinal(), value);
}

@Override
public Vector2D clone() {
    return new Vector2D(getX(), getY());
}

@Override
public String toString() {
    return vec.toString();
}

@Override
public void add(Vector2D other) {
    vec.add(other.vec);
}

@Override
public void sub(Vector2D other) {
    vec.sub(other.vec);
}

@Override
public void multiply(Vector2D other) {
    vec.multiply(other.vec);
}
```

```
@Override
public void multiply(double value) {
    vec.multiply(value);
}

@Override
public void divide(Vector2D other) {
    vec.divide(other.vec);
}

@Override
public void divide(double value) {
    vec.divide(value);
}

@Override
public double getSize() {
    return vec.getSize();
}

@Override
public double getSizeSquared() {
    return vec.getSizeSquared();
}

@Override
public double getDotProduct(Vector2D other) {
    return vec.getDotProduct(other.vec);
}

@Override
public double getDistance(Vector2D other) {
```

```
    return vec.getDistance(other.vec);
}

@Override
public Vector2D getForwardVector() {
    var forwardVec = clone();
    forwardVec.vec.set(forwardVec.vec.getForwardVector());
    return forwardVec;
}

@Override
public double getAt(int index) {
    return vec.getAt(index);
}

@Override
public void setAt(int index, double value) {
    vec.setAt(index, value);
}

@Override
public Vector2D getOpposite() {
    var v = clone();
    v.toOpposite();
    return v;
}

@Override
public void toOpposite() {
    vec.toOpposite();
}
```

```
    public void rotate(double degrees) {
        var x = getX();
        var y = getY();
        var rad = Math.toRadians(degrees);
        var coss = Math.cos(rad);
        var sinn = Math.sin(rad);
        setX(x * coss - y * sinn);
        setY(x * sinn + y * coss);
    }
}
```